

1991

# Reducing Computational Expense of Ray-Tracing Using Surface Oriented Pre-Computation

Robert E. Rinker

*University of North Florida*



# REDUCING COMPUTATIONAL EXPENSE OF RAY-TRACING USING SURFACE ORIENTED PRE-COMPUTATION

by

Robert E. Rinker

A thesis submitted to the
College of Computer and Information Sciences
in partial fulfillment of the requirements for the degree of

Master of Science in Computer and Information Sciences

UNIVERSITY OF NORTH FLORIDA
COLLEGE OF COMPUTER AND INFORMATION SCIENCES

April, 1991

The thesis "Reducing Computational Expense of Ray-Tracing Using Surface Oriented Pre-Computation" submitted by Robert E. Rinker in partial fulfillment of the requirements for the degree of Master of Science in Computer and Information Sciences has been

Approved by the thesis committee: Date

Signature Deleted

April 29, 1991

Thesis advisor and Committee Chairman

Signature Deleted

April 29 1991

Signature Deleted

April 29 1991

Accepted for the College of Computer and Information Sciences:

Signature Deleted

April 29 1991

Interim Dean

Accepted for the University:

Signature Deleted

April 29, 1991

Vice-President for Academic Affairs

CONTENTS

List of Figures and Tables . . . . . v

Abstract . . . . . vi

Chapter 1: Introduction . . . . . 1

1.1 Computational Expense . . . . . 2

1.2 Purpose . . . . . 6

Chapter 2: The Pre-Computational Method . . . . . 7

2.1 The Applicable Scenes . . . . . 8

2.2 The Computations . . . . . 9

Chapter 3: Implementation and Testing . . . . . 11

3.1 Generating the Test Scenes . . . . . 11

3.2 The Pre-Computation Routine . . . . . 12

3.3 Testing Procedures . . . . . 13

Chapter 4: Results . . . . . 15

4.1 Statistical Analysis Technique . . . . . 16

4.2 Analysis of Results . . . . . 17

4.3 Conclusions . . . . . 18

References . . . . . 20

Appendix A: Source Code for Random Patch Generator . . . . . 22

Appendix B: Source Code for Pre-Computation Routine . . . . . 24

Appendix C: Source Code for Rendering Routine . . . . . 29

Appendix D: Computational Details . . . . . 40

Appendix E: Sample Rendered Scene . . . . . 43

Vita . . . . . 44

FIGURES AND TABLES

Figure 1: Tracing A Ray Through A Scene . . . . . 2

Figure 2: Viewpoint From Point In Scene . . . . . 8

Table 1: Intersection Results . . . . . 15

Table 2: Time Results . . . . . 15

## ABSTRACT

The technique of rendering a scene using the method of ray-tracing is known to produce excellent graphic quality, but is also generally computationally expensive. Most of this computation involves determining intersections between objects in the scene and ray projections. Previous work to reduce this expense has been directed towards ray oriented optimization techniques. This paper presents a different approach, one that bases pre-computation on the characteristics of the scene itself, making the results independent of the position of the observer. This means that the results of one pre-computation run can be applied to renderings of the scene from multiple view points. Using this method on a scene of random triangular planar patches, impressive reductions in the number of intersection computations was realized, along with significant, reductions in the time required to render the scene.

# Chapter 1

# INTRODUCTION

The problem of effectively and accurately rendering 3- dimensional scenes on a 2-dimensional display screen has been addressed since the beginning of modern computer graphic manipulation. One technique that has enjoyed widespread popularity is ray tracing (sometimes referred to as ray casting) [Roth82]. This technique is based on the same physical model that physicists use in tracing light rays through lenses, and was first proposed for computer graphics rendition in the 1950's [Glassner89]. The technique basically uses the idea that the computer view screen is in fact not a continuum, but is composed of a finite number of discrete points, or pixels. Using basic geometric computations, and the elementary physical laws of reflection and refraction, individual rays of "light" can be traced from the view point, through each pixel in the view screen, and into the scene. Using this "backward tracing" technique, only those rays that are actually perceived by the viewer need be considered. In general, as each of these rays intersects a surface in the scene, it will be partially reflected, partially refracted, and partially absorbed, the proportions depending on the nature of the surface. The point of intersection, then, can be thought of as the origin of up to two "child" rays, each of which must be traced to a possible intersection with a surface in the scene, which may then be the origin of up to two additional children. At some point in the process, the spawned rays must be deemed to have terminated, by exiting the scene, by an arbitrary limit

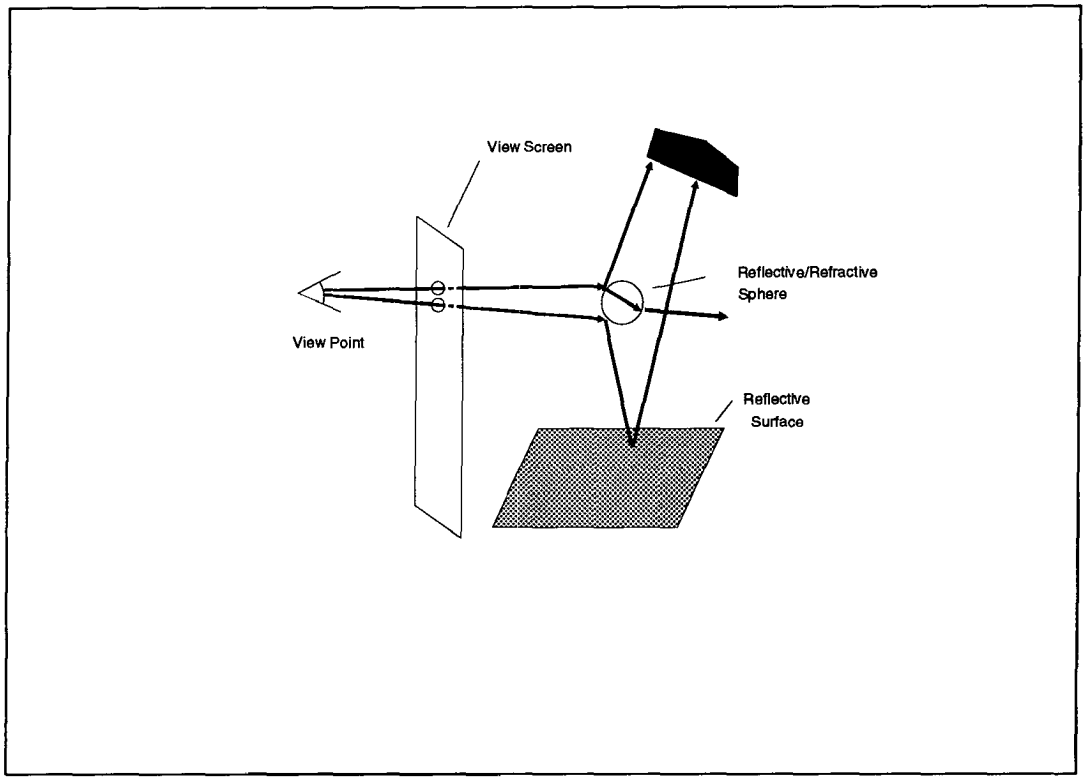


Figure 1: Tracing a Ray Through A Scene.

on the number of generations considered, or by the ray's intensity having attrited below some designated threshold. The resultant color and intensity assigned to the view screen pixel will be an appropriate combination of the characteristics of all of the surface points that the ray and its children intersected. The one dimensional ray, is then used to determine the attribute of the 2 dimensional pixel area.

## 1.1 Computational Expense

While the basic theory of ray tracing is very simple, it is apparent that intense computation is necessary to actually implement the process. For that reason, the use of this technique had to wait for the hardware to catch up (about 1970), before it enjoyed a resurgence of interest from the initial ideas of the 1950's. In particular,

the problem of computing the proper intersections of the rays with the surfaces in the scene requires a significant expenditure of computational resources. Compounding the expense is the fact that it must be done in floating point, with a good degree of precision. Whitted [Whitted80] determined that 75% of the computational time involved in rendering a scene using ray tracing involved computing these points of intersection. It is apparent, then, that efforts in speeding up ray tracing renderings should be most fruitful in this area. Kaplan [Kaplan85], listed several properties that speed-up schemes should exhibit; essentially noting that they should be relatively independent of scene composition or complexity, require a "reasonable" amount of time to execute (in the context of pre-computation), be independent of the direction of a ray, and be amenable to parallelism. Basically, three general strategies have been followed [Arvo89]: reducing the average computational costs of ray-surface intersections, reducing the number of intersections which must be computed for a particular ray, and replacing individual rays with aggregate structures (beams).

Much work has been done in the area of reducing the average computational costs of individual intersections. Two branches of this idea have developed - bounding volume techniques [Whitted80][Roth82], and hierarchical scene space subdivision techniques[Glassner84][Glassner88] -as well as combinations of the two.[Sweeney86] Both techniques seek to reduce the candidate surfaces that must be tested for intersection, by representing them by volumes for which intersection computations are simpler - typically spheres, parallelepipeds, or other regular solids. Myriad variations on this idea have been explored, including hierarchical bounding volumes and combinations of various bounding shapes. The payoff of these techniques is that

if the ray in question does not intersect a particular bounding volume, any surfaces or volumes wholly contained within this bounding volume need not be tested for intersection. Of course, if the bounding volume IS intersected, then all other volumes and surfaces it contains must also be tested, and the computation that determined the intersection with the parent volume was wasted. This "overhead" is acceptable, however, since the cost of the bounding volume intersection calculation is relatively small, and the return in terms of surfaces that can be omitted if the volume is NOT intersected is large.

Glassner's approach was to partition the scene space into non-uniform cubic volumes (sometimes called voxels)[Glassner84]. He started with a cube that encompassed the entire scene space, and whose edges were parallel with the major axes of the scene. The cube was divided into 8 identical non-overlapping sub cubes, each also oriented with the major axes. Each sub cube was examined to see if it contained any surfaces in the scene, or if it was entirely contained within some object in the scene. If the cube did not contain any parts of any objects, or was entirely contained within an object, it was not sub-divided any further. However, if the cube DID contain all or part of any object in the scene, it itself was divided into 8 equal non-overlapping sub cubes. The process continued until either no cube contained more than one object, or a pre determined depth of sub cubes had been reached. As the family of sub cubes was generated, an "octree" was constructed to provide rapid access to any cube in the structure. The root of the octree was the original cube, and each generation was represented by a level of the tree. The resultant structure was used as follows: As a ray was spawned or entered the scene (in the case of the first generation rays

from the viewer), it was tested for intersection with any object in the voxel it was in. If there was no intersection found, the ray was advanced in increments equal to the length of the edge of the smallest voxel in the structure, until it was in a different voxel. Intersections were then tested within this voxel. If none were found the ray was again advanced. This process continued until the ray reached the edge of the scene, or it intersected an object. In testing the intersections within a particular voxel, bounding volumes were used to further reduce the computational expense of the technique. Glassner reported significant speed up with this technique over "naive" ray tracing methods [Glassner84]. Variations on this method involved different tree structures, different methods of subdividing the scene space, and different bounding volume construction techniques.

The techniques developed to increase the speed of ray tracing, other than the fairly basic device of spherical bounding volumes, seem to not adequately address the needs in animation, or even to be particularly useful in rendering the same scene from different viewpoints. The main stumbling block is that if the view point changes, the display axes change. That means that all of the advantage gained by orienting our bounding volumes or space divisions with the major axes has disappeared. In fact it now becomes nearly as expensive to test these contrived objects as to test the original objects in the scene. The obvious remedy is to re-orient the developed structures so that they are aligned with the new axes. But this requires that the whole process start from the beginning again. It would seem to be useful to have available a method in which the pre-computation would only be done once, and then the attributes so derived could be used to advantage in rendering the

scene from ANY viewpoint. It seems obvious that any strategy that would meet that goal must be based on internal characteristics of the scene, rather than external ones (such as axes).

## 1.2 Purpose

The purpose of this research was to devise a technique for pre-computing the characteristics of a scene to be ray traced based on the internal characteristics of the scene, rather than the characteristics and paths of individual rays travelling through the scene. The results of the pre-computation were to be useable for rendering the scene as it would appear from any viewer position, without recalculation being necessary if that position were to change. The technique was to be applicable to scenes of various levels of complexity, as measured by the number of different surfaces in the scene, and should produce a net reduction in computational costs to render the scene. That is, the cost of the pre-computation was not to exceed the reduction in rendering realized by using the pre-computed results.

Chapter 2

# THE PRE-COMPUTATION METHOD

All of the speed up methods that have been discussed are what can be referred to as "ray based" methods. That is, they are oriented around a particular ray, tracking it through it's life. The technique to be discussed and used here is "scene based." The pre-computation involved describes the characteristics of the scene, independently of any ray paths through it. The basic approach is to try to answer to as great an extent possible, the question of what can be "seen" from a particular point in a scene. It is important to note that this approach will not provide any information to help in computing intersections for rays that originate from outside the scene. In particular, the rays "originating" from the viewpoint will not gain any computational advantage from the pre-computed information. But for all the "offspring" rays (generated by reflection and refraction) of these original rays, the pre-computation will provide the information that from a particular point in the scene only a subset of the surfaces in the scene are visible. Therefore, the candidates for intersection by that offspring ray can be limited. That is to say, if a ray is reflected from that point, the daughter ray need only be tested for intersection with this subset of the surfaces in the scene. At first thought, this might seem like an impossible task, since there is an infinite number of points in any scene. Any attempt to limit the number of points to some discrete set will certainly result in gaps in the scene and such things as ragged edges. However, not all points in the scene are important. It is only necessary to consider

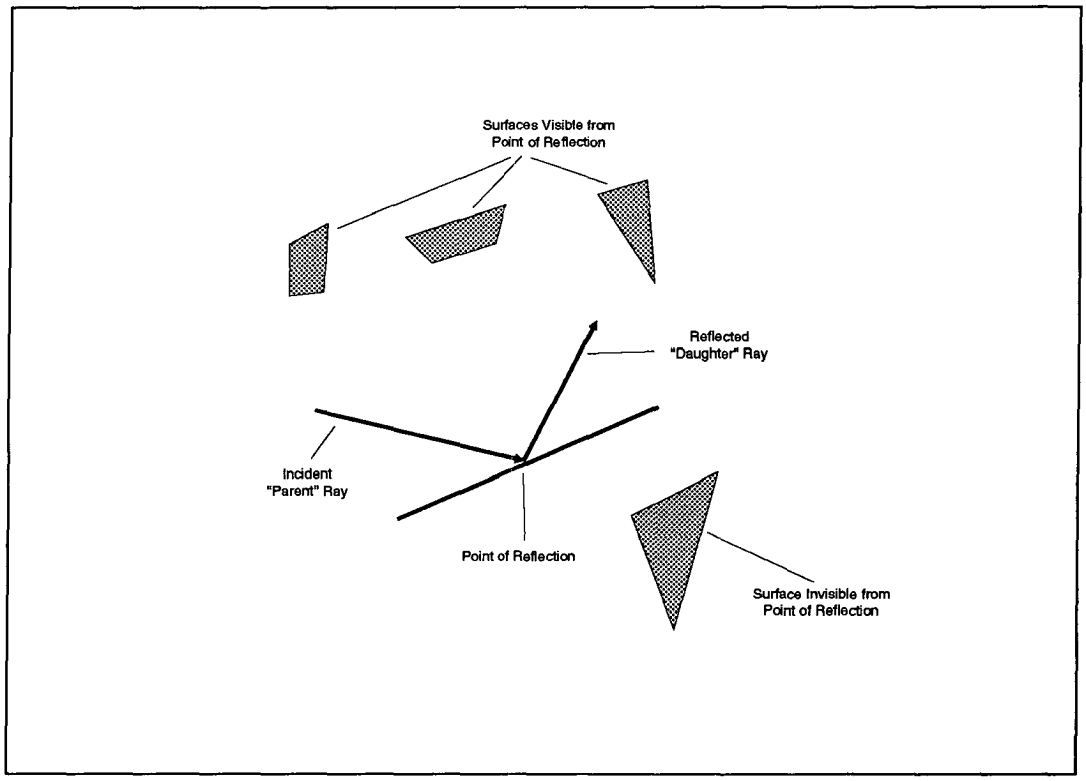


Figure 2: Viewpoint from point in scene.

points that are part of surfaces in the scene, since these are the only points that can be the origin of reflected rays. With this thought, the set of points to be considered has been reduced by a large amount, however the set remaining is still non-finite!

## 2.1 The Applicable Scenes

If the scene contents are limited slightly, the problem can be simplified immensely. In particular, if the scene is limited to one containing surfaces that are composed of planar patches, certain characteristics of all points on a particular patch can be generalized by examining the points that make up the vertices of the hull of the patch - certainly a finite set! Since representing objects with collections of planar patches is a widely used technique in computer graphics, [Watt89] this should not be

a serious limitation. We also will not require that the patches be convex, but simply that they be planar.

## 2.2 The Computations

The actual pre-computation is done in three phases. The idea is to reduce the set of possible candidates for intersection by a ray reflected from the surface in question in each phase. At the end, a list of only those other surfaces in the scene that could possibly be seen from the surface in question is associated with that surface. This list is highly robust in that the scene can be rotated or translated, or the observer can move and the pre-computed information is still valid. The only thing that can not be done is to move the surfaces in the scene in relation to each other.

For a particular surface, each of the other surfaces in the scene is first examined to see if it lies entirely behind the surface, where it would be invisible to a ray emanating from the front of the surface. Those that do lie behind the surface, are eliminated from the list of candidates. Next, each of the remaining candidates are examined to see if their front surfaces are turned away from the front surface of the surface being examined. Any for which this is determined to be the case are also eliminated from the list. Finally, the hemisphere visible from the surface in question is divided into overlapping solid sectors, each sector being that region that is contained within a right circular cone constructed with its vertex on the surface, and its axis parallel to the normal to the surface. The sectors are defined by cones with different interior angles. Each of the candidates are examined to determine if they are

completely within one of these sectors. A candidate that is found to be in one of the sectors is not eliminated, but the fact is included in the pre-computation information. This categorization is used to eliminate this candidate for a particular ray being traced if the angle between the ray and the normal to the parent surface is not smaller than the interior angle of the cone defining the sector containing the candidate. The result of the pre-computation is a list for every surface in the scene of those other surfaces that are visible from its perspective. It is important to note that this approach, by its very nature, will not yield any information as to rays that originate from outside the scene. Specifically, the initial ray from the viewer through the view screen will not gain any information from the pre-computation.

# Chapter 3

# IMPLEMENTATION AND TESTING

There is a great danger in testing a particular graphic technique, that the scenes selected will have characteristics that will influence the results of the test. To avoid this, the scenes used to test the pre-computation technique consisted of a randomly generated collection of triangular planar patches. The patches were allowed to intersect each other, and were not restricted to their placement or orientation in the scene. The patches were, however, restricted to the maximum size they could be. This was done to prevent the scene from being blocked by one large patch, and to ensure an adequate level of complexity of the scene. The scene itself was a 256x256x256 unit cube. This size was selected to allow adequate room to place the patches while at the same time keeping the scene small enough to allow test runs to complete in a reasonable time.

## 3.1 Generating the Test Scenes

The patches were generated by first using a pseudo random number generator to generate 3 triples between 0 and 256, inclusive. These were, of course, considered coordinates of the vertices of the patch. The length of the edges between adjacent vertices was computed and, if any edge was longer than 100 units, two of the

vertices were discarded and re-generated. This process continued until a patch was obtained that had no edge longer than 100 units.

## 3.2 The Pre-Computation Routine

The pre-computation routine performed the calculations by first considering each of the planes defined to have a front and a back. The surface was assumed invisible from the back. The front was determined by the order of the vertices - to an observer on the front side of the plane the vertices were listed in counter clockwise order. Each surface was then considered in order to develop a description of the scene from the viewpoint of the front of that surface. First, each of the other patches was tested to see if all three of its vertices were on the back side of the plane. If they were, that patch was "marked" invisible and not considered further. Of the remaining candidates, each was tested to see if it was perhaps turned in such a way that its front surface was not visible from the surface of the patch in question. Since these both were areas and not points, the determination was based on the lines of sight between a point on the candidate patch and each of the vertices of the patch in question. If the angles between these lines of sight and the normal to the candidate patch were all greater than 90 degrees, the candidate surface was determined to be invisible from anywhere on the patch. Finally, each candidate patch was tested to determine if it was contained in one of the angular sectors described above. Again, the fact that we were dealing with surfaces and not with points made the computation less straight forward than it might have otherwise been. The procedure involved generating three right circular cones, each with its vertex at a vertex of the

patch, with axes parallel to the normal to the plane of the patch, and with interior angles equal to the angle of the sector in question. If a candidate was found to have all 3 vertices contained within all three of these cones, the candidate was deemed to be contained within the sector. The results of the pre-computation steps were stored in a structure associated with each of the surfaces in the scene. Within each of these structures, 32 bits (4 bytes) were used to contain the information on each patch in the scene. So in the case of a 100 patch scene, for example, there would be 100 of these structures, each being 400 bytes in size, for a total of 40K of storage - hardly an imposition on any reasonable system.

## 3.3 Testing Procedures

A rendering program was written that allowed a choice of using or not using the pre-computed data. If the choice was mad to not use the pre computation data, the program rendered the scene using a naive rendering technique - simply checking each of the surfaces for an intersection for each reflection of a ray. If the pre-computed data was used, only those surfaces that were in the visible list, and which were not precluded because of being in the wrong angle sector, were checked for intersection. A depth of reflection of 5 was selected for the test runs. This seemed to give an adequate level of ray spawning and in practice resulted in most of the rays being reflected out of the scene. A few runs with depths of reflection of 8 were made, but no significant differences in results were noted, most probably due to most rays having exited the scene by 5 reflections. In the pre-computation, eight angle sectors were used to categorize the candidates. This number was entirely arbitrary,

and seemed to be a reasonably fine division. The system was tested using scenes containing 2, 25, 50, 75, 100, 150, and 200 patches. Testing was done on a NeXt computer using the MACH operating system. Some testing was done on a personal computer running an Intel 80386 processor, and on one using an Intel 80486 processor, both using the MS-DOS operating system. Other than a different speed of execution, no significant differences in results were noted on the different platforms. For each level of scene complexity (number of surfaces in the scene), 30 independent sets of patches were generated and each scene was rendered with and without using the pre-computed information. Data was taken as to the number of intersection computations that had to be made, as well as the total system time required to render the scene. The time was measured using the Unix Korn shell time utility. The times reported are the sum of system time and user time. Thirty independent replications of each situation were performed to make valid statistical conclusions easy to reach.

Chapter 4

Results

Table 1 summarizes the results of the test runs. Each entry in the table is given in the form mean/standard deviation. As would be expected, very simple scenes, represented by a small number of surfaces, gain very little from pre-computation. As the complexity of the scene increases, the effectiveness of the pre-computation also increases. The results are particularly dramatic in terms of a reduction in the number of intersections that must be computed, reaching nearly 60% in the case of 200 surfaces. The time advantages are not so dramatic but are still significant, exceeding 20% in the case of 200 surfaces.

| Summary Of Results | | | |
|---|---|---|---|
| INTERSECTION COMPUTATIONS REQUIRED | | | |
| Number of Surfaces | Without Pre-computation | With Pre-computation | Differences |
| 2 | 297/404 | 82/265 | 214/360 |
| 25 | 89037/35649 | 34958/17697 | 54079/20905 |
| 50 | 348156/73878 | 141315/49012 | 206841/52133 |
| 75 | 892486/145301 | 381501/91246 | 510984/85647 |
| 100 | 1480248/171255 | 633038/140948 | 847209/96561 |
| 150 | 3407729/306824 | 1500592/201350 | 1907137/196847 |
| 200 | 6008754/611312 | 2586560/406000 | 3422194/322911 |

Table I: Intersection Results

| Summary Of Results | | | |
|---|---|---|---|
| TIME REQUIRED (seconds) | | | |
| Number of Surfaces | Without Pre-computation | With Pre-computation | Differences |
| 2 | 27/11 | 26/11 | 1/2 |
| 25 | 262/55 | 250/51 | 12/8 |
| 50 | 588/72 | 510/65 | 47/14 |
| 75 | 943/94 | 831/79 | 112/22 |
| 100 | 1315/112 | 1129/103 | 185/31 |
| 150 | 2302/127 | 1888/98 | 414/56 |
| 200 | 3439/247 | 2687/209 | 751/90 |

Table II: Time results

4.1 Statistical Analysis Technique

The results of the test runs were statistically analyzed using a test-of-hypothesis technique to test for differences in the number of intersection computations and in the time needed to render the scene with and without using the pre-computed information. For each scene, the null hypotheses were that there was no difference in the average number of intersection computations that were made, nor was there any difference in the time required to render the scene. The alternative hypotheses were that the number of intersection calculations and the time required were decreased by using the pre-computed information. Since the sample sizes in all cases were 30, a Z score was used as the test statistic, and the population variances were estimated using the sample variances. A Z score of 1.65 was used as a critical value. This yields results at a significance level of 95%.

## 4.2 Analysis of Results

At even the lowest level of complexity (2 surfaces), a significant difference in the number of intersection computations was realized, the data yielding a Z score of 2.42. The Z score for the difference in number of intersection computations continued to increase as the complexity (number of surfaces) increased. The differences in time did not become statistically significant until the scene contained 50 surfaces, when the Z score became 2.66. Again all Z scores for more complex scenes were higher than this. Adding to the viability of this method is the fact that the expense of performing the pre-computation is practically nil. The most complex scene took less than 5 seconds to process through the pre-computation on the NeXt machine. Essentially the method resulted in a 60% reduction in computational requirements for very little computational expense! The storage requirements for the pre-computed information were also very modest, the largest file being only 169K bytes (for 200 surfaces). The disparity between the surface intersection reduction and time reduction was examined closely. Tests were run using profile monitors to account for the time the rendering program was taking. It was found that nearly 40% of the total system time was used to compute the first ray intersection for rays entering the scene from the observer, the remaining 60% being used to compute subsequent ray-surface intersections. Since the pre-computation technique purposefully does not address the first ray intersection, the results obtained are not surprising.

## 4.3 Conclusions

The method of scene oriented pre-calculation is a viable and inexpensive method of reducing the computational expense of rendering complex scenes using ray tracing. It appears that as the scene becomes more complex, the pre-computation becomes more valuable. This method would be particularly useful in repeated rendering of the same scene from different viewpoints, as might be done in animation. It is felt that the reduction in intersection computations is a better measure of the worth of the method than is the gross time measurements, for several reasons. First, the main objective of pre-computation, by this or any of the methods mentioned in chapter one, is the reduction of intersection calculations. Directly measuring this by examining the differences in number of intersection calculations required with and without using the pre-computation seems the best way to decide if the method accomplishes that objective. Second, the time measurement could be directly affected by clever coding techniques (or the lack of them), as well as the efficiency of the compiler, both clouding the central issue of computational expense reduction. For example, two methods of scene rendering might be essentially performing the same underlying algorithm, but because one was inefficiently coded, or because the compiler did not optimize it's source code well, it could take significantly longer to execute.

Particularly since by it's very nature, this pre-computation technique cannot address rays entering the scene from the outside, it would seem that, as is so often the case, the best results would probably be realized by combining this technique with one or more of the others mentioned in the introduction. It would appear that both bounding

volumes and hierarchical scene space subdivision could be combined with the method presented here and yield improved results, particularly in the reduction of time of execution. This would seem to be a promising avenue for further research.

## REFERENCES


[Arvo89]
Arvo, J. and Kirk, D. "A Survey of Ray Tracing Acceleration Techniques", An Introduction to Ray Tracing, edited by A. Glassner,Academic Press, 1989, 201-262.

[Glassner84]
Glassner, A. "Space Subdivision for Fast Ray Tracing", IEEE Computer Graphics and Applications, v4(10), Oct 1984, 15-22.

[Glassner88]
Glassner, A. "Spacetime Ray Tracing for Animation", IEEE Computer Graphics and Applications, March 1988,p60-70.

[Glassner89]
Glassner, A.S. "An overview of Ray Tracing", An Introduction to Ray Tracing, edited by A. Glassner, Academic Press, 1989, p1-31.

[Goldsmith87]
Goldsmith, J. Salmon, J., "Automatic Creation of Object Hierarchies for Ray Tracing", IEEE Computer Graphics Applications, May 1987, 14-20.

[Kaplan85]
Kaplan, M.R. "Space Tracing, a Constant time Ray Tracer" Computer Graphics (proceedings SIGGRAPH 85) seminar notes from "State of the Art in Image Synthesis", July 1985.

[Roth82]
Roth, S.D. "Ray Casting for Modeling Solids", Computer Graphics and Image Processing, v18, 1982 p109-144.

[Sweeney86]
Sweeney, M.A.J., Bartels, R.H., "Ray Tracing Free Form B-Spline Surfaces", IEEE Computer Graphics and Applications, Feb 1986, 41-49.

[Watt89]
Watt, Alan, Fundamentals of Three-Dimensional Computer Graphics Addison-Wesley, 1989, p31.

[Whitted80]
Whitted, T. "An improved Illumination Model for Shaded Display", Communications of the ACM, v23(6), June 1980, p343-349.

Appendix A

SOURCE CODE LISTING FOR RANDOM PATCH GENERATOR

```
/*Random patch Generator. Accepts command line or interactive input to determine
number of patches to be generated. Randomly generates patches with no edges larger
than a specified length*/

#include <stdlib.h>
#include <time.h>
#include <stdio.h>
#define MAX_INT 0x7FFFFFFF
#define FNAME    "surfaces.bin"   /*output file*/

struct surface
   {float verts[3][3];} s ;    /*holds vertex coordinates*/
#define MAXDIST  (long)100*100  /*edge limit*/
#define DIMENSION  256          /*max coordinate value*/

/*macro to generate 3 scaled random coordinates*/
#define GENERATE(x)  for(j=0;j<3;j++)\
                              s.verts[(x)][j]=(float)random()/\
                               RAND_MAX *DIMENSION

#define SQR(x)   ((long)x)*((long)x)

main(int argc, char *argv[])
{int number,i=0,j;
 long dist1,dist2,dist3;
 FILE *outfile;
 if (argc>1)                /*check to see if argument was*/
   sscanf(argv[1],"%d",&number);   /*supplied on cmd line*/
 else                       /*if not, get interactively*/
    {printf("How many surfaces are to be generated? ");
     scanf("%d",&number);
    }
 if( (outfile=fopen(FNAME,"wb"))==NULL)
   {printf("Cannot open output file, surfaces.bin\n");
    exit (1);
   }
```

```
 fwrite(&number,sizeof(int),1,outfile);  /*write number of
surfaces generated to output file*/
 srandom((int)(time(0)%MAX_INT));  /*randomize generator*/

 for (;i<number;i++)      /*generate the vertices*/
   { dist1=dist2=dist3=MAXDIST+1;
     GENERATE(0);      /*generate 1st vertex*/
     s.verts[0][2]=-s.verts[0][2];
    while(dist1>MAXDIST||dist2>MAXDIST||dist3>MAXDIST)
      {  GENERATE(1);    /*then next two*/
         GENERATE(2);
         s.verts[1][2]=-s.verts[1][2];  /*adjust Z coords for right
                                          hand coordinate system*/
         s.verts[2][2]=-s.verts[2][2];
  /*compute edge lengths and check them*/
         dist1=SQR(s.verts[0][0]-s.verts[1][0]);
         dist1+=SQR(s.verts[0][1]-s.verts[1][1]);
         dist1+=SQR(s.verts[0][2]-s.verts[1][2]);
         if(dist1>MAXDIST) continue;
         dist2=SQR(s.verts[0][0]-s.verts[2][0]);
         dist2+=SQR(s.verts[0][1]-s.verts[2][1]);
         dist2+=SQR(s.verts[0][2]-s.verts[2][2]);
         if (dist2>MAXDIST) continue;
         dist3=SQR(s.verts[1][0]-s.verts[2][0]);
         dist3+=SQR(s.verts[1][1]-s.verts[2][1]);
         dist3+=SQR(s.verts[1][2]-s.verts[2][2]);
      }  /*here everything is ok, so we write the
surface*/
    fwrite(&s,sizeof(struct surface),1,outfile);
  } /*end of for loop*/
 fclose(outfile);
}
```

Appendix B

SOURCE CODE LISTING FOR PRECOMPUTATION ROUTINE

```
/*Precomputation program. Reads a file of surface vertex coordinates. Performs the
precomputation and writes a file to be used by the rendering program*/

#include<stdio.h>
#include<math.h>
#define SQR(x)    (x)*(x)
#define FNAME   "surfaces.bin"   /*input file name*/
#define ONAME "output.bin"       /*output file name*/
#define INTERSECT  0x80000000   /*bit mask to indicate
                                 intersecting surface*/
#define VISIBLE    0x40000000   /*bit mask to indicate
                                 visible surface*/
#define INVISIBLE  0
#define NOANGLES    8    /*number of angle sectors to use*/
typedef int FOURBYTES;            /*Whatever locl type is 4 bytes*/

typedef struct            /*structure to hold info for surfaces*/
  {float normal[3];       /*Direction #s of normal to surface*/
   float verts[3][3];    /*coordinates of surface vertices*/
   FOURBYTES *vis;       /*array of flags for other surfaces in the scene*/
  }ss;

int how_many;        /*number of surfaces to be processed*/
ss *s_array;        /*array of surface structures*/

float *x_prod(float *, float *);
float *coeffs(float *,float);
float angle_tans[NOANGLES-1];   /*actual angles used for angle sector definition*/

main()
{int i,noa,j;
 FILE *ofile;
 read_surfs();                          /*read input file*/
 ofile=fopen(ONAME,"wb");
 fwrite(&how_many,sizeof(int),1,ofile);   /*write number of
                                          surfaces to follow to output file*/
 for(i=0;i<how_many;i++)     /*for each surface do three
                                                  precomputation steps*/
{compute_hidden(i,s_array[i].normal[0],s_array[i].normal[1],
```

```
            s_array[i].normal[2]);
    compute_back(i,s_array[i].verts);
    compute_cones(i);
              /*and write results to file*/
    fwrite((s_array+i)->normal,sizeof(float),3,ofile);
    for(j=0;j<3;j++)
      fwrite(&s_array[i].verts[j],sizeof(float),3,ofile);

/*the structure contains a pointer to the array of 4 byte flags - we have to write the
actual flags following each structure*/

fwrite((s_array+i)->vis,sizeof(FOURBYTES),how_many,ofile);
  } /*end of for loop*/

 noa=NOANGLES-1;   /*write actual sector angles*/
 fwrite(&noa,sizeof(int),1,ofile);
 fwrite(angle_tans,sizeof(float),noa,ofile);
 fclose(ofile);
 }

/**************************COMPUTE_CONES******************
Accepts the index of a surface. Computes all those other surfaces that are all in an
angle division from the surface in question.
*/

compute_cones(int surface)
{FOURBYTES mask=0;   /*this is the angle flag mask*/
 int inside,i,j,k,k1;
 float *cc,po2,ang,delang;
 po2=asin(1.L);            /*1/2 pi*/
 delang=po2/NOANGLES;   /*angle increment between sectors*/
 for(i=1;i<NOANGLES;i++) /*process each surface for each sector*/
   {ang=po2-i*delang;
    if (ang<0.L) ang=0.1;
    cc=coeffs(s_array[surface].normal,ang);
    angle_tans[i-1]=cc[6];
    for (j=0;j<how_many;j++)
/*this if checks that surface hasn't already been eliminated, and that it is not in
another sector by checking its flags*/
if((s_array[surface].vis[j]&VISIBLE)&&!(s_array[surface].vis[j]&mask))
      /*now see if it's in THIS sector*/
       {for(k=0;k<3;k++)
         {for(k1=0;k1<3;k1++)
           {inside=inside1(s_array[j].verts[k1],cc,
                s_array[surface].verts[k]);
            if (!inside) break;
           }
```

```
          if(!inside) break;
         }
        if (inside)
/*if it is, set the appropriate bit flag*/
s_array[surface].vis[j]=s_array[surface].vis[j]|(1L<<(i-1));
        mask=(mask<<1)+1;  /*shift mask*/
       }
   }

 }

/*****************************COMPUTE_BACK*******************
```
accepts the index of a surface structure and the vertices that define the surface. Goes through all other visibility flags, and tests to see if the front surfaces of those that are visible can be seen from any of the vertices of the surface in question. If it cannot, that visibility flag is set to 0.

```
*/
compute_back(int surface,float verts[][3])
{int i=0,k,j;
 float vi,vj,vk,dot;
 for (;i<how_many;i++)  /*go through the surfaces*/
   {if (s_array[surface].vis[i]&VISIBLE)  /*if a surface has not already been marked
                                            as invisible*/
     {for(k=0;k<3;k++)   /*check it from all vertices*/
       {vi=verts[k][0]-s_array[i].verts[0][0];
        vj=verts[k][1]-s_array[i].verts[0][1];
        vk=verts[k][2]-s_array[i].verts[0][2];
        dot=s_array[i].normal[0]*vi+s_array[i].normal[1]*vj+
            s_array[i].normal[2]*vk;
        if (dot>0) break;
       }
  /*if we get here, it cannot be seen from any of the three vertices*/
        if (dot<=0)  s_array[surface].vis[i]=INVISIBLE;
     }
 /*now check the intersecting surfaces in the same manner*/
   if(s_array[surface].vis[i]&INTERSECT)
     {for(k=0;k<3;k++)
       {for(j=0;j<3;j++)
        {vi=verts[k][0]-s_array[i].verts[j][0];
         vj=verts[k][1]-s_array[i].verts[j][1];
         vk=verts[k][2]-s_array[i].verts[j][2];
         dot=s_array[i].normal[0]*vi+s_array[i].normal[1]*vj+
             s_array[i].normal[2]*vk;
         if(dot>0) break;
        }
        if(dot>0)break;
       }
```

```
        if (dot<=0) s_array[surface].vis[i]=INVISIBLE;
       }
    }
  }

/*****************************COMPUTE_HIDDEN*****************
Accepts the index of a surface structure and its normal direction cosines. Goes
through all visibility flags in the surface, sets those representing surfaces that are
behind the plane of the subject surface to zero, and those that intersect the plane of
the given surface to INTERSECT.
*/

compute_hidden(surface,si,sj,sk)
int surface;
float si,sj,sk;
{float front,sum,on,v[3];
 int i,j,rise[3];
 for(i=0;i<3;i++)
    v[i]=s_array[surface].verts[0][i];
/*using equation of plane, decide what values are in front and which values are
behind*/
 on=v[0]*si+v[1]*sj+v[2]*sk;
 front=(v[0]+si)*si;
 front+=(v[1]+sj)*sj;
 front+=(v[2]+sk)*sk;
 front=front-on;
 for(i=0;i<how_many;i++)
   if(s_array[surface].vis[i]&VISIBLE)
     {for(j=0;j<3;j++)
/*now check all vertices of each surface*/
{sum=s_array[i].verts[j][0]*si+s_array[i].verts[j][1]*sj+
                s_array[i].verts[j][2]*sk-on;
        if(fabs(front+sum)==(fabs(front)+fabs(sum)))
          rise[j]=1;
        else
          rise[j]=0;
        }
     if( (rise[0]==rise[1])&&(rise[1]==rise[2]))
        {if (!rise[0])
          s_array[surface].vis[i]=INVISIBLE;
        }
     else  /*if any are not on the same side*/
       s_array[surface].vis[i]=INTERSECT;
     }
 }

/*****************************READ_SURFS*******************
```

```
Opens the file of surface vertices and reads them into the surface structure array.
Initializes structure arrays and computes surface unit normals.
*/

read_surfs()
{FILE *infile=NULL;
 int number,i;
 if ( (infile=fopen(FNAME,"rb"))==NULL)
  { printf("Cannot open file %s for input\n",FNAME);
   exit (1);
  }
 fread(&number,sizeof(int),1,infile);
 if ((s_array=(ss *)malloc(number*sizeof(ss)))==NULL)
   {printf
("Error in allocating memory for surface array\n");
    fclose(infile);
    exit (0);
   }
 how_many=number;
 for(i=0;i<number;i++)
 fread((s_array+i)->verts,9*sizeof(float),1,infile);
 fclose(infile);
 {int k;
  float *temp,len,v1[3],v2[3];
  for(i=0;i<number;i++)
  {s_array[i].vis=
(FOURBYTES *)malloc(how_many*sizeof(FOURBYTES));
  /*this sets all visibility flags to "visible" except the flag representing the surface
itself*/
   for(k=0;k<how_many;k++)
     s_array[i].vis[k]=VISIBLE;
   s_array[i].vis[i]=INVISIBLE;
   for(k=0;k<3;k++) /*This computes the unit normal to the
surface*/
    {v1[k]=s_array[i].verts[1][k]-s_array[i].verts[0][k];
     v2[k]=s_array[i].verts[2][k]-s_array[i].verts[0][k];
    }
   temp=x_prod(v1,v2);
   len=sqrt(SQR(temp[0])+SQR(temp[1])+SQR(temp[2]));
   for(k=0;k<3;k++)
      s_array[i].normal[k]=temp[k]/len;
  }
 }
}
```

Appendix C

SOURCE CODE FOR RENDERING ROUTINE

```
/*Rendering program. Accepts an option to use or not use precomputed information.
OPens and reads either the file of surface vertices or the output file from the
precomputation program. Traces rays throught the scene*/

#include<stdio.h>
#include<math.h>
#include<time.h>
#define SQ(x) (x)*(x)
#define MAXDEPTH 5        /*depth of reflections*/
#define VIEWX 128      /*viewer coordinates*/
#define VIEWY 128      /*(right hand system)*/
#define VIEWZ 500
#define GENFILE "surfaces.bin"   /*input file of vertices*/
#define PREFILE "output.bin"    /*file from pre-computation
   routine*/
#define PRE 1
#define NOPRE 0
#define HEIGHT 256      /*scene dimensions*/
#define WIDTH 256
typedef long FOURBYTES;
typedef struct {
  float normal[3];
  float verts[3][3];
  FOURBYTES *vis;
  } ss;
ss *s_array;   /*array of surface structures*/

float intercept_pt(float *, float *, int, float *);
/*Computes the intersecvtion of a ray from a given point with a given set if
direection numbers with a particular surface*/

int inside(float *,int);
/*Given a point on the plane of a sutrface, and a surface, determines if the point lies
within the bounds of the surface*/

void reflect(int , float *, float *);
/*given an incident ray and a surface, computes the direction numbers of the
reflected ray*/
```

```
int bounce (int,int,float *,float *,int);
/*A recursive routine that tracks the ray through it's lifetime and returns the number
of the surface it ends at, or a -1 if the ray exits the scene*/

int first_hit(int,int,float *, float *);
/*computes the first point of intersection of a ray entering the scene, with a surface
in the scene*/

float *x_prod(float *, float *);
/*computes the cross product of two vectors*/

void no_pre(void);
void pre(void);
/*these two routines drive the ray tracing, either without (no_per) or with (pre)
precomputed information*/

void read_surfs(void);
void read_surfs2(void);
/*These routines open and read the appropriate files into the program (read_surfs2 if
precomputed information is to be used)*/

float dot(float *, float *);
/*computes the dot product bewteen two vectors*/

int check_angle(FOURBYTES, float);
/*checks if a particular surface flls inside a praticular angle sector or not*/

int how_many;       /*number of surfaces in the scene*/
int noangles;      /*number of angle sectors*/
float *angle_tans;  /*actual tangents of angle sectors*/
unsigned long number_checked=0;  /*performance measurement variable - counts
number of surface intersections computed*/

main(int argc , char *argv[])
{char ans[80];
 time_t start_time, end_time;
 double e_time;
 if (argc>1)
   {ans[0]=toupper(argv[1][0]);}
 else
   ans[0]='\0';
 while ((ans[0]!='Y')&& (ans[0]!='N'))
   {
    printf("Use Pre computation? (Y/N) :");
    gets(ans);
    ans[0]=toupper(ans[0]);
   }
```

```
  time(&start_time);
  if (ans[0]=='N')
    no_pre();
  else
    pre();
  printf("%d surfaces to depth %d\n",how_many, MAXDEPTH);
  printf("Number checked = %ld",number_checked);
  time(&end_time);
  e_time=difftime(end_time,start_time);
  printf("\n%c  Elapsed time = %.0lf\n",ans[0],e_time);

 }

/**********************INTERCEPT_PT**********************
function that accepts the coordinates of a starting point (x1,y1,z1), a set of direction
numbers (i,j,k), and the index number of a surface. It computes the intersection of a
line through the starting point in the direction given with the plane containing the
surface. The coordinates (x,y,z) of the point of intersection are returned in the last
parameter, and the linear parameter is returned by the function. If there is no
intersection, a -1 is returned as the linear parameter. Computations based on the
intersection of the line: x=x1+it, y=y1+jt, z=z1+kt, with the plane a(x-x2)+b(y-y2) +
c(z-z2)=0.
*/

float intercept_pt(start_pt, direction, surface,
returned_pt)
float start_pt[], direction[], returned_pt[];
int surface;
{float x1,y1,z1,x2,y2,z2,i,j,k,a,b,c;
 double t,num,den;
 x1=start_pt[0];
 y1=start_pt[1];
 z1=start_pt[2];
 i=direction[0];
 j=direction[1];
 k=direction[2];
 a=s_array[surface].normal[0];
 b=s_array[surface].normal[1];
 c=s_array[surface].normal[2];
 x2=s_array[surface].verts[0][0];
 y2=s_array[surface].verts[0][1];
 z2=s_array[surface].verts[0][2];
 den=a*i+b*j+c*k;
 if (den==0)
   return -1.;
 num=a*(x2-x1)+b*(y2-y1)+c*(z2-z1);
 t=num/den;                    /*compute parameter*/
```

```
 if (t<0)                    /*if its neg, its behind us*/
    return -1.;
 returned_pt[0]=x1+i*t;   /*compute coords of intersection*/
 returned_pt[1]=y1+j*t;
 returned_pt[2]=z1+k*t;
 return t;
 }

 /*************************INSIDE*****************************
 Accepts the coordinates of a point, and a surface index. Returns a 1 if the point is in
the bounds of the surface, a zero otherwise.
*/

 int inside(float pt[],int surface)
 {float maxx=0, maxy=0, minx=HEIGHT, miny=WIDTH;
  float maxz=0,minz=HEIGHT;
  double verts[3][3],normal[3],v[3][3],dot,sum,angle;
  int i,j;
  const float two_pi=4*asin(1.);
/*make sure point is in scene*/
  if(pt[0]<0 || pt[1]<0 || pt[2]>0) return 0;
  if(pt[0]>HEIGHT || pt[1]>WIDTH || pt[2]<-HEIGHT) return 0;
  for (i=0;i<3;i++)
   {for(j=0;j<3;j++)
      verts[i][j]=s_array[surface].verts[i][j];
    normal[i]=s_array[surface].normal[i];
   }
  for(i=0;i<3;i++)       /*find x,y,z bounds of patch*/
    {if (verts[i][0]>maxx) maxx=verts[i][0];
     if (verts[i][0]<minx) minx=verts[i][0];
     if (verts[i][1]>maxy) maxy=verts[i][1];
     if (verts[i][1]<miny) miny=verts[i][1];
     if (verts[i][2]<minz) minz=verts[i][2];
     if (verts[i][2]>maxz) maxz=verts[i][2];
    }
  minx-=.01;    /*adjustments for pts on edges*/
  maxx+=.01;
  miny-=.01;
  maxy+=.01;
  maxz+=.01;
  minz-=.01;
  if (pt[0]<minx||pt[0]>maxx) return 0;
  if (pt[1]<miny||pt[1]>maxy) return 0;
  if (pt[2]<minz||pt[2]>maxz) return 0;
  for (i=0;i<3;i++)         /*compute vectors from point*/
   for(j=0;j<3;j++)              /*to vertices*/
     v[i][j]=verts[i][j]-pt[j];
```

```
  for (i=0;i<3;i++)          /*normalizing vectors*/
    {sum=0;
     for (j=0;j<3;j++)
      sum+=v[i][j]*v[i][j];
     sum=sqrt(sum);
     for (j=0;j<3;j++)
       v[i][j]=v[i][j]/sum;
     }
/*now see if sum of angles from point to vertices is 360 deg*/
  dot=v[0][0]*v[1][0]+v[0][1]*v[1][1]+v[0][2]*v[1][2];
  angle=acos(dot);
  dot=v[1][0]*v[2][0]+v[1][1]*v[2][1]+v[1][2]*v[2][2];
  angle+=acos(dot);
  dot=v[2][0]*v[0][0]+v[2][1]*v[0][1]+v[2][2]*v[0][2];
  angle+=acos(dot);
  if (fabs(angle-two_pi)<.1) return 1;
  else return 0;
}

/************************REFLECT*****************************
given a surface index and a direction of incidence, returns the direction numbers of
the reflected ray using the formula (Reflected direction)  = (incident direction) +
2(normal to surface)cos(angle between incident ray and normal to surface)
*/
void reflect(int surface, float direction[], float new_direction[])
{float val_dir,costh=0., dir[3];
 int i;
 for(i=0;i<3;i++)
    {dir[i]=direction[i];
    costh+=s_array[surface].normal[i]* -dir[i];
   }
 val_dir=0.;
 for(i=0;i<3;i++)
   {new_direction[i]=dir[i]+2*s_array[surface].normal[i]*costh;
    val_dir+=SQ(new_direction[i]);
   }
 val_dir=sqrt(val_dir);
 for  (i=0;i<3;i++)
   new_direction[i]=new_direction[i]/val_dir;
 }



/**********************BOUNCE*****************************
accepts a depth integer, the coordinates of the current point, the direction numbers of
the current ray, the index of the current surface, and a flag indicating whether
pre-computed information is to be used. Recursively tracks the ray through depth
```

```
levels of reflection and returns the index number of the surface at the termination of
the ray. If the ray leaves the scene, a -1 is returned.
*/

int bounce (depth, surface, pt, direction,pre_nopre)
float pt[], direction[];
int depth,surface,pre_nopre;
{int i,hot_surface;
 float t=1e100,t1;
 float returned_pt[3],rp[3],new_direction[3],dotp;
/*end of recursion*/
 if (depth>=MAXDEPTH) return surface;

 for (i=0;i<how_many;i++)  /*start looking for next intersection*/
   if (s_array[surface].vis[i]&&
          ((dotp=dot(s_array[i].normal,direction))<0)&&
(!pre_nopre||check_angle(s_array[surface].vis[i],dotp)))
     {t1=intercept_pt(pt,direction,i,returned_pt);
     number_checked++;
      if(t1>0&&t1<t && inside(returned_pt,i))
        {int j;   /*find closest patch intersected*/
         t=t1;
         hot_surface=i;
         for(j=0;j<3;j++)
              rp[j]=returned_pt[j];
        }
      }
 if (t<1e100)   /*means we found one*/
    {reflect(hot_surface,direction,new_direction);
    return bounce(depth+1,hot_surface,rp,new_direction,pre_nopre);
    }
 return -1;       /*means we left scene*/
 }

 /************************FIRST_HIT*************************
 accepts x and y coordinates on view screen. Computes ray initial vector and
intersects it with surface hit. Returns surface hit, point of intersection, and direction
numbers of reflected ray. If nothing is hit, returns -1.
 */

 int first_hit(x,y,pt, direction)
 int x,y;
 float pt[],direction[];
 {float i,j,k,len,sp[3],dir[3],ip[3],sum;
  int surface,ii,ij;
  float t=1e100,t1;
  i=x-VIEWX;
```

```
 j=y-VIEWY;
 k=-VIEWZ;
 len=sqrt(SQ(i)+SQ(j)+SQ(k));
 dir[0]=i/len;
 dir[1]=j/len;
 dir[2]=k/len;
 sp[0]=x;
 sp[1]=y;
 sp[2]=0.;
 for (ii=0;ii<how_many;ii++)
   {sum=0;
    for(ij=0;ij<3;ij++)
      sum+=dir[ij]*s_array[ii].normal[ij];
    if(sum<0)
     {t1=intercept_pt(sp,dir,ii,ip);
      if (t1<t&&inside(ip,ii))
        {surface=ii;
         for(ij=0;ij<3;ij++)
            pt[ij]=ip[ij];
         t=t1;
        }
     }
    }
 if (t<1e100)
   {reflect(surface,dir,direction);
    return surface;
   }
 return -1;
}

/*********************NO_PRE*********************************
Function called to render scene if no pre calculations are used
*/
void no_pre()
{int i,j,surface;
 float ip[3], direction[3];
 read_surfs();
 for(i=0;i<HEIGHT;i++)
  {
   for(j=0;j<WIDTH;j++)
     {
      if ((surface=first_hit(i,j,ip,direction))>-1)
       {
        surface=bounce(0,surface,ip,direction,NOPRE);
       }
     }
   }
```

```
  }

/************************READ_SURFS************************
Opens the file of surfaces and reads them into the surface structure array. Initializes
structure arrays and computes surface unit normals. This is the file that contains only
the surface vertices - it contains no precomputed information.
*/

void read_surfs()
{FILE *infile=NULL;
 int number,i;
 if ( (infile=fopen(GENFILE,"rb"))==NULL)
  { printf("Cannot open file %s for input\n",GENFILE);
   exit (1);
  }
 fread(&number,sizeof(int),1,infile);
 if ((s_array=(ss *)malloc(number*sizeof(ss)))==NULL)
   {printf("Error in allocating memory for surface array\n");
    fclose(infile);
    exit (0);
   }
 how_many=number;
 for(i=0;i<number;i++)
 fread((s_array+i)->verts,9*sizeof(float),1,infile);
 fclose(infile);
 {int k;
  float *temp,len,v1[3],v2[3];
  for(i=0;i<number;i++)
  {   for(k=0;k<3;k++)
    {v1[k]=s_array[i].verts[1][k]-s_array[i].verts[0][k];
     v2[k]=s_array[i].verts[2][k]-s_array[i].verts[0][k];
    }
   temp=x_prod(v1,v2);
   len=sqrt(SQ(temp[0])+SQ(temp[1])+SQ(temp[2]));
   for(k=0;k<3;k++)
      s_array[i].normal[k]=temp[k]/len;
   s_array[i].vis=
      (FOURBYTES *)malloc(sizeof(FOURBYTES)*how_many);
   for(k=0;k<how_many;k++)
      if(k!=i)
        s_array[i].vis[k]=1;
      else
        s_array[i].vis[k]=0;
  }
 }
}
```

```
/*******************X_PROD*********************************
accepts two 3 dimensional floating point vectors and returns their floating point cross
product.
*/
float *x_prod(float *a, float *b)
{static float xp[3];
 xp[0]=a[1]*b[2]-a[2]*b[1];
 xp[1]=a[2]*b[0]-a[0]*b[2];
 xp[2]=a[0]*b[1]-a[1]*b[0];
 return xp;
}

/************************PRE*********************************
called to render using precomputed information
*/
pre()
{int i,j,surface;
 float ip[3],direction[3];
 read_surfs2();
 for(i=0;i<HEIGHT;i++)
   {
    for(j=0;j<WIDTH;j++)
    {
     if ((surface=first_hit(i,j,ip,direction))>-1)
        surface=bounce(0,surface,ip,direction,PRE);
    }
   }
 }

/********************DOT***********************************
accepts 2 3-d vectors and returns their dot product, normalized
*/
float dot(float *a, float *b)
{float sum=0,sumsq=0;
 int i;
 for(i=0;i<3;i++)
    {sum+=a[i]*b[i];
     sumsq+=sum*sum;
    }
if(sumsq==0)
  return 0;
 sum=sum/sqrt(sumsq);
 return sum;
 }

 /************************READ_SURFS2************************
 reads the precomputed information into the surface arrays
```

```
 */

read_surfs2()
{FILE *infile;
 int number,i;
 if ( (infile=fopen(PREFILE,"rb"))==NULL)
  { printf("Cannot open file %s for input\n",PREFILE);
   exit (1);
  }
 fread(&number,sizeof(int),1,infile);
 if ((s_array=(ss *)malloc(number*sizeof(ss)))==NULL)
   {printf("Error in allocating memory for surface array\n");
    fclose(infile);
    exit (0);
   }
 how_many=number;
 for(i=0;i<how_many;i++)
  {fread(s_array[i].normal,sizeof(float),3,infile);
   fread(s_array[i].verts,sizeof(float),9,infile);
   s_array[i].vis=
       (FOURBYTES *)malloc(how_many*sizeof(FOURBYTES));
   fread(s_array[i].vis,sizeof(FOURBYTES),how_many,infile);
  }
  fread(&noangles,sizeof(int),1,infile);
  angle_tans=(float *)malloc(sizeof(float)*noangles);
  fread(angle_tans,sizeof(float),noangles,infile);
}

/*******************CHECK_ANGLE*****************************
accepts a visibility string, a surface index and the cosine of an angle. Determines if
surface flag is set for the appropriate angle.
*/

check_angle(FOURBYTES vis, float costh)
{float tanth;
 int i;
 FOURBYTES mask=1;

 tanth=tan(acos(costh));
 if(!(0x3FFFFFFF&vis)) return 1;
 for (i=0;i<sizeof(FOURBYTES)*8-1;i++)
   {if (mask&vis)
     {if (tanth>angle_tans[i])
        return 1;
      else
        return 0;
      }
```

```
    mask=mask<<1;
  }
}
```

Appendix D

COMPUTATIONAL DETAILS

The pre-computation routine accepts a structure of vertex coordinates, three vertices to a surface. It first computes the unit normal to the plane containing the surface, by calculating the vector cross product between a vector from the first vertex to the second, and one from the first to the third, and then normalizing the result. This vector establishes the "front" and "back" sides of the surface, the vector being considered to point in the direction of the "front". Using these direction numbers $<N_x,N_y,N_z>$ and the coordinates of the first vertex $(X_1,Y_1,Z_1)$, the equation of the plane is found to be:

$$<N_x,N_y,N_z><X-X_1,Y-Y_1,Z-Z_1>=0$$

Next, the coordinates of a point one unit in front of the surface are found by sampling moving in the direction of the unit normal from the first vertex. These coordinates are substituted in the left side of the equation to establish the sign resulting when points that are "in front of" the surface are substituted into the equation. Points that are on the surface will yield a result of zero, and points that are "behind" the surface will yield a result of the opposite sign as those points "in front of" the surface. Once this has been established, the coordinates of each of the vertices of all other surfaces in the scene are substituted into the left side of the plane equation and the resulting signs are tested using the fact that $|A| + |B| =$

$|A + B|$ only if A and B have the same sign. If a surface is found to have all three of its vertices "behind" the surface in question, it is marked as invisible from that surface. The surfaces remaining are then tested to see if they are oriented in such a way that their front sides are not visible to the testing surface. This is done by defining vectors from each vertex of the testing surface to a single vertex of the tested surface, and computing the vector dot products between each of these vectors and the unit normal of the surface being tested. If any of the dot products are negative, the conclusion is that the vertex on the surface being tested is visible from at least one point on the testing surface. Otherwise, the conclusion is that that vertex, and hence all other points on the planar surface being tested, is invisible from all points on the testing surface, and the tested surface will be marked as invisible from the testing surface. Sets of right circular cones are next defined. The cones are constructed with their axes parallel to the unit normal of the testing surface. This is computed by starting with a right circular cone whose vertex is at the origin and whose major axes is the Z axes. The equation of that cone will be:

$$\frac{X^2}{A^2}+\frac{Y^2}{A^2}-\frac{Z^2}{C^2}=0$$

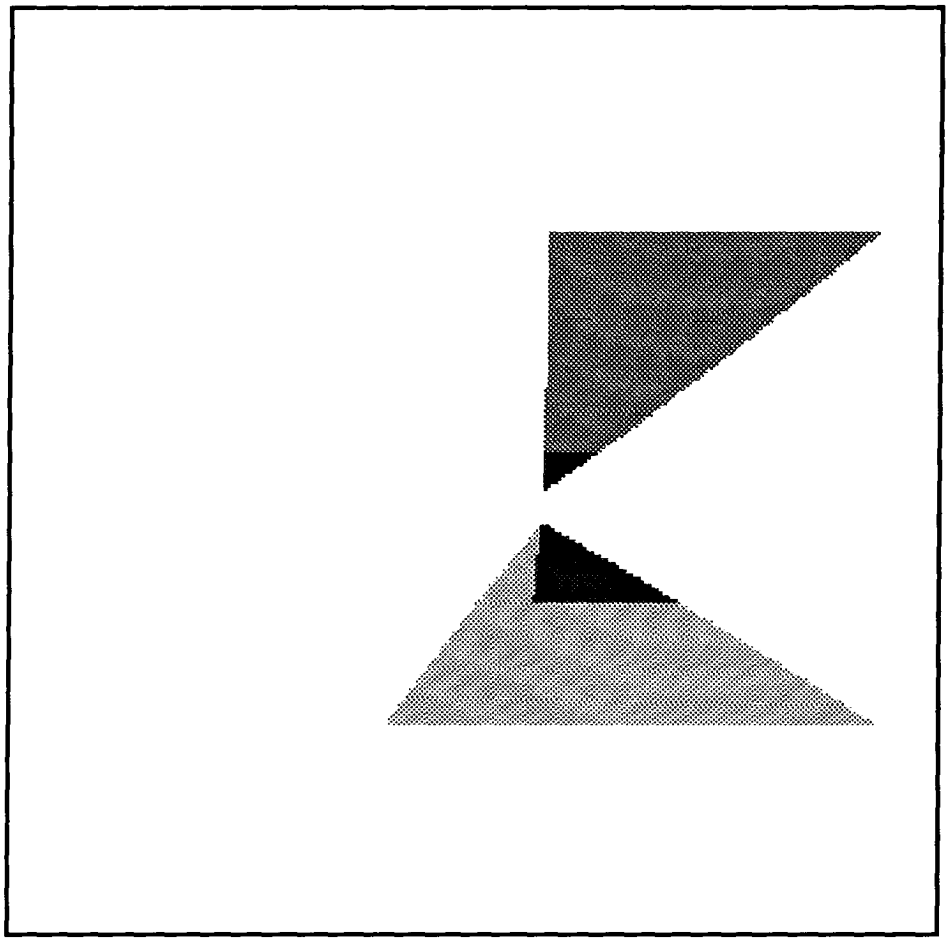
where C/A is the tangent of the interior angle between the axis and the surface of the cone. If the coordinate axis system is rotated so that the new axes have direction numbers relative to the original axis system of $<X_i,X_j,X_k>$, $<Y_i,Y_j,Y_k>$, and $<Z_i,Z_j,Z_k>$, then the equation of the cone, as it is reoriented to remain parallel with the new Z axis, can be found by substituting in the above equation the three scaler products, $X<X_i,Y_i,Z_i>$, $Y<X_j,Y_j,Z_j>$, and $Z<X_k,Y_k,Z_k>$, for X, Y and Z respectively.

After squaring the resulting numerators and gathering terms, the equation of the newly oriented cone is found to be: $(X_i^2/A^2 + X_j^2/A^2 - X_k^2/C^2)X^2 + (Y_i^2/A^2 + Y_j^2/A^2 - Y_k^2/C^2)Y^2 + (Z_i^2/A^2 + Z_j^2/A^2 - Z_k^2/C^2)Y^2 + 2(X_iY_i/A^2 + X_jY_j/A^2 - X_kY_k/C^2)XY + 2(X_iZ_i/A^2 + X_jZ_j/A^2 - X_kZ_k/C^2)XZ + 2(Y_iZ_i/A^2 + Y_jZ_j/A^2 - Y_kZ_k/C^2)YZ = 0.$ Finally, translating the resultant cone so that it's vertex is at a point in space with coordinates $X_1,Y_1,Z_1$, requires only substituting $(X-X_1)$, $(Y-Y_1)$, and $(Z-Z_1)$ for X,Y and Z in the above equation.

During the pre-computation, a set of cones is generated for each of the required sector angle divisions, computed by simply dividing 90 degrees by the desired number of equal segments, then setting the interior angle of the appropriate set of cones equal to a multiple of the product. So, for example, if 3 divisions were desired, one set would be generated with interior angles equal to 30 degrees, and one set with angles equal to 60 degrees. A set of these cones would consist of 3 cones, all parallel to the normal to the testing surface, one from each vertex of the testing surface. After each set is generated, using the above equations, all of the other surfaces in the scene are tested to see if all of their vertices lie entirely within the three cones of the set. This test is done by substituting the coordinates of the vertices into a cone equation and determining if the result is positive or negative. If it is negative, the vertex is inside that particular cone. If all of the vertices of a surface are inside a set of cones, the surface is considered to be visible only to a ray whose angle of incidence with the testing surface is greater than or equal to the interior angle of the set of cones. This information is coded into the output file of pre-computed information.

Appendix E

Sample Rendered Scene



Example scene with 2 surfaces

# VITA

Robert E. Rinker has a Bachelors degree from the United States Naval Academy, at Annapolis, Md., 1964, and a Master of Science degree in Operations Research and Systems Analysis from the United States Naval Postgraduate School, at Monterey, Ca., 1971. He expects to receive a Master of Science degree in Computer and Information Systems from the University of North Florida in May 1991. Dr. Yap Chua of the University of North Florida is serving as Robert's thesis advisor.

Robert retired from a career as a Naval Aviator in 1984, and is currently employed as a Professor of Mathematics at Florida Community College at Jacksonville, Florida. He has been interested in computers and computer applications since 1962, and has written applications in myriad languages, including C, Pascal, Fortran, Pl/1, Algol, Cobol, Prolog, Lisp, Simscript, and GPSS, both in and out of the academic environment.